Compiler construction

Lecture 7: Functions

Alex Gerdes
Vårtermin 2016

Chalmers University of Technology — Gothenburg University

Nested functions

A Nested Function

Suppose we extend JAVALETTE with nested functions.

```
double hypSq(double a, double b) {
  double square(double d) {
    return d * d;
  }
  return square(a) + square(b);
}
```

Another example

To make nested functions useful we would like to have lexical scoping.

This means that we can use variables in the inner function, defined in the outer function.

```
double sqrt(double s) {
  double newton(double y) {
    return (y + s / y) / 2;
  }
  double x = 0.0; int i = 0;
  while (i < 10) {
    x = newton(x);
    i++;
  }
  return x;
}
```

Access links

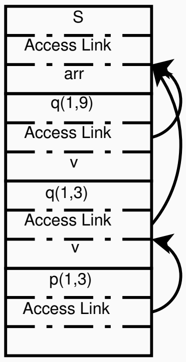
- Access links are a mechanism to access variables defined in an enclosing procedure
- An access link is an extra field in a stack frame which points to the closes stack frame of the enclosing procedure
- An access link is sometimes called a static link

Access links

Outline of a quicksort implementation:

```
void sort(int[] arr) {
  void quicksort(int m, int n) {
    int v = ...
    void partition(int y, int z) {
      write(y);
      ... arr ... v ...
    }
    ... arr ... v ... partition ... quicksort
  }
  void write(int x) {
    ... arr ... x ...
  }
  ... quicksort ...
}
```

Example stack



When accessing e.g. the variable `arr` in `p` we need to go through the access link to `q` and then to `s`.

Following access links

When procedure `q` calls procedure `p` there are three cases to consider:

1. `p` has higher nesting depth than `q`

Then the depth of `p` must be exactly one larger than `q` and `p`'s access link must point to `q`.

2. `p` and `q` have the same nesting depth

The access link for `p` is the same as for `q`.

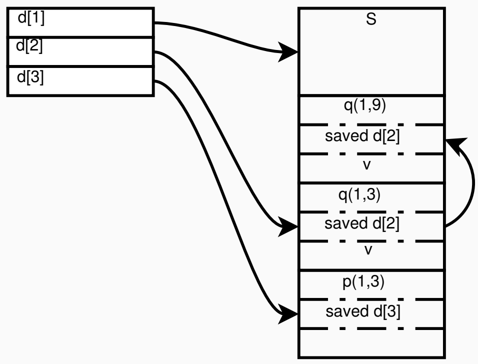
3. `p` has a lower nesting depth than `q`

Let n_p be the nesting depth of `p` and n_q be the nesting depth of `q`. Furthermore, suppose that `p` is defined immediately within procedure `r`. The top activation record for `r` can be found by following $n_q - n_p + 1$ access links up the stack.

Displays

- If the nesting depth is very large, then the link chains may be very long; traversing these links can be costly
- Displays were developed to speed up access
- A Display is an stack, separate from the call stack, which maintains pointers to the most recent activation record of the different nesting depths
- The display grows and shrinks with the maximum nesting depth of the functions on the call stack

Displays



Lambda lifting

- Another way of implementing nested functions is by lifting them to the top level
- Free variables are handled by adding them as parameters to the lifted function

Lambda lifting - example

Original `sqrt`

```
double sqrt(double s) {
  double newton(double y) {
    return (y + s / y) / 2;
  }
  double x = 0.0;
  int i = 0;
  while (i < 10) {
    x = newton(x);
  }
  return x;
}
```

Lambda lifting - example

Lambda lifted `sqrt`

```
double newton(double y, double s) {
  return (y + s / y) / 2;
}

double sqrt(double s) {
  double x = 0.0;
  int i = 0;
  while (i < 10) {
    x = newton(x,s);
  }
  return x;
}
```

Another lifting example

Consider lambda lifting the function below.

```
void foo() {
  int c = 0;
  void incc() {
    c++;
  }
  incc();
  incc();
  printInt(c);
}
```

Call-by-reference

The local function `incc` modifies its free variable. In order to lift `incc` we have to pass the parameter `c` by reference.

```
void incc(int *c) {
  (*c)++;
}
void foo() {
  int c = 0;
  incc(&c);
  incc(&c);
  printInt(c);
}
```

Higher order functions

Higher order fuctions in JAVALETTE

Adding higher order functions to JAVALETTE we need a new form of types:

```
Type(Type, ..., Type)
```

Examples:

- `bool(int, int)`
  A function which takes two `int` arguments and returns a `bool`
- `void()`
  A function which takes no arguments and doesn't return anything

Higher order functions in JAVALETTE

```
int main() {
  int(int) add(int n) {
    int h(int m) {
      return n + m;
    }
    return h;
  }

  int(int) addFive = add(5);
  printInt(addFive(15));
}
```

Higher order functions in JavaLette

```
int main() {
  int(int) add(int n) { ... }
  int(int) addFive = add(5);

  int(int) twice(int(int) f) {
    int g(int x) {
      return f(f(x));
    }
    return g;
  }

  int(int) addTen = twice(addFive);
  printInt(twice(twice(addTen))(2));
}
```

Implementing higher order functions

There are several ways implementing higher order functions:

- Access Links can be adapted to also deal with higher order functions
- Defunctionalization is a method to convert higher order functions to data structures; requires whole program compilation
- Closures are used to represent functions by a heap allocated record containing a code pointer and the free variables of the function
- Using closures is by far the most common implementation method

Defunctionalization example[1]

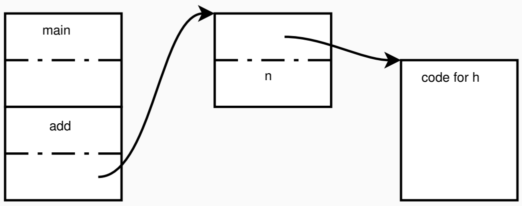
```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)

cons :: a -> [a] -> [a]
cons x xs = x : xs

o :: (b -> c) -> (a -> b) -> a -> c
o f g x = f (g x)

walk :: Tree t -> [t] -> [t]
walk (Leaf x)     = cons x
walk (Node t1 t2) = o (walk t1) (walk t2)

flatten :: Tree t -> [t]
flatten t = walk t []
```

[1] Adapted from Olivier Danvy / Wikipedia

Defunctionalization example

```
data Lam a = LamCons a
           | LamO (Lam a) (Lam a)
           | LamWalk (Lam a)

apply :: Lam a -> [a] -> [a]
apply (LamCons x)  xs = x : xs
apply (LamO f1 f2) xs = apply f1 (apply f2 xs)
apply (LamWalk f)  xs = apply f xs

cons_def :: a -> Lam a
cons_def x  = LamCons x

o_def :: Lam a -> Lam a -> Lam a
o_def f1 f2 = LamO f1 f2

walk_def :: Tree t -> Lam t
walk_def (Leaf x)     = LamWalk (cons_def x)
walk_def (Node t1 t2) = LamWalk (o_def (walk_def t1) (walk_def t2))

flatten_def :: Tree t -> [t]
flatten_def t = apply (walk_def t) []
```

Closures



- A closures is a pair of a code pointer and an environment containing the free variables of the function
- The closure for `h` inside `add` contains a pointer to the code for `h` and the value for the variable `n`
- The closure is heap allocated

Closures and mutable variables

What happens with the stack allocated variable `counter` once we exit the function `makeCounter`?

```
int() makeCounter(int start) {
  int counter = start;
  int inc() {
    counter++;
    return counter;
  }
  return inc;
}
```

Closures and mutable variables

What happens with the stack allocated variable `counter` once we exit the function `makeCounter`?

- Heap allocate part of the stack frame
- Forbid such programs (example: Java)

```
int() makeCounter(int start) {
  int counter = start;
  int inc() {
    counter++;
    return counter;
  }
  return inc;
}
```

Closures and mutable variables

Functional languages like Haskell and ML deal with the problem of closures and mutability as follows:

- Everything is immutable by default
- Mutation is introduced by references which always live on the heap

```
makeCounter = do
  r <- newIORef 0
  let inc = do
    n <- readIORef r
    writeIORef r (n+1)
    return n
  return inc
```

Anonymous nested functions

Lambda expressions

- An increasingly popular language feature is to have anonymous nested functions, so called lambda expressions
- Compiling lambda expressions works the same way as nested functions with names

A note on terminology

- One can often hear the phrase that a language “has closures”
- This is a somewhat unfortunate use of the word
- Closures is an implementation technique for the language feature higher order functions

Lazy evaluation

Question

- Is it possible to implement `if` as a function?

Question

- Is it possible to implement `if` as a function?
- We can fake it by using functions which take no arguments

```
void if(bool c, void() th) {
  if (c)
    th();
}
```

- We emulate lazy evaluation with this construct

Example - lazy lists

```
typedef struct Node *lazylist;

struct Node {
  int elem;
  lazylist() next;
}

lazylist cons(int x, lazylist() xs) {
  list res = new Node;
  res->elem = x;
  res->next = xs;
  return res;
}

int sum(lazylist xs) {
  if (xs == (lazylist)null)
    return 0;
  else
    return xs->elem + sum(xs->next());
}
```

Example - lazy lists

```
int main() {
  printInt(sum(take(42, enumFrom(1))));
  return 0;
}

lazylist enumFrom(int n) {
  lazylist rec() { return enumFrom(n + 1); }
  return cons(n, rec);
}

lazylist take(int n, lazylist xs) {
  if (xs == (lazylist)null)
    return xs;
  else if (n < 1)
    return (lazylist)null;
  else {
    lazylist rec() { return take(n - 1, xs->next()); }
    return cons(xs->elem, rec);
  }
}
```

Thunks

- Call-by-name is a calling convention where the arguments are not evaluated until needed
- Thunks are used to implement call-by-name
- Thunks are essentially functions which take no arguments
- They are typically implemented as closures

Lazy evaluation

- The difference between call-by-name and lazy evaluation is that once an argument is evaluated, it is not reevaluated if it is used twice
- In order to achieve laziness, once the value is computed we need to remember it
  This can be done in two ways:
  - Overwrite the thunk with an indirection pointing to the value
  - Overwrite the thunk with the value directly, if the space allocated for the thunk is big enough to hold the value

A Note

- Call-by-name and lazy evaluation is very handy as they allow the programmer to create new control structures
- Be careful with combining them with side-effects: it can yield very surprising results
- An impure language with lazy evaluation as default is a bad idea